AUTODESK UNIVERSITY

MFG124959

Vault Extensions Deep Dive

Explore Vault Extension and Automation Programming

Christian Gessner
coolOrange S.r.l.

Markus Koechl
Autodesk – Central Europe

Learning Objectives

- Understand the details of the Vault API, and learn how to create your own functionality and extend existing Vault features
- Learn how to create and debug own Vault Explorer extensions, jobs, and event handlers
- Learn how to create your own Vault Explorer extensions, your own jobs, and your own event handlers to automate workflows and repetitive tasks
- Learn how to use Vault functionalities from within own applications or from within custom Inventor add-ins

Description

Vault data management software isn't just a single program; it's a framework, composed of many pieces like CAD add-in or Vault Explorer (Client) working together. Some of these pieces are customizable and some are not. This class will discuss the different capabilities of the Vault API, in particular, details of Vault Explorer extensions, custom jobs, and custom event handlers – including aspects like debugging, security, and other sophisticated topics.

AUTODESK UNIVERSITY

Your AU Experts

Christian Gessner is a co-founder and a software engineer at coolOrange. He is involved in the design, specification and implementation of data management projects and customizations, in particular in the Autodesk data management environment. Besides that, he teaches programming and visits customers for on-site trainings and workshops. Previously, Christian worked as a SQA engineer for data management products at Autodesk.

Markus Koechl studied mechanical engineering and science of work. Since starting at Autodesk in 1999, he has continuously contributed to customers' success implementing 3D CAD and PDM solutions for more than 15 years. He also has experience in implementation, customization, and automation of CAD and PDM solutions products of Dassault SolidWorks. In his current role as solutions engineer for PDM/PLM, Markus covers Vault software and all of its interfaces, extensions and link to Fusion Lifecycle. He helps Autodesk resellers, prospects, and customers to analyze and optimize workflows and practical approaches in the management of engineering data and downstream processes.

AUTODESK UNIVERSITY

Contents

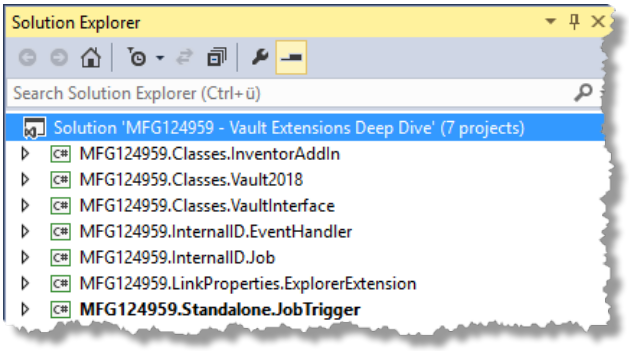
1 Introduction

Read this handout to get an overview of the different possibilities to extend and automate Vault. Based on 3 real world use cases, some selected and sophisticated examples are provided which can either be used to learn Vault's programming environment or put to work as extended features in your Vault environment.

1.1 Target Audience

This class primarily addresses programmers but is also suitable for administrators, consultants and applications engineers experienced in .NET coding with both basic and advanced knowledge of the Vault API.

1.2 Sample Materials



A Visual Studio solution with all use cases described in this class is shared in the ***MFG124959.zip*** archive in the folder ***Visual Studio Class Solution***.

All the projects are written in C#.

To compile the solution, the Vault SDK for Vault 2018 has to be installed to the default location:

C:\Program Files (x86)\Autodesk\Autodesk Vault 2018 SDK.

If the SDK is installed to a different location, the references to the Vault assemblies have to be changed in all the particular projects.

The vault environment used by the authors to create code projects and documentation is shared in the ***Vault Backup – Sample Vault SD124422-MFG124959.zip*** archive.

To restore the downloaded backup, follow the instructions in Vault Help applying all default options. Autodesk Data Management Server Console - Restore.

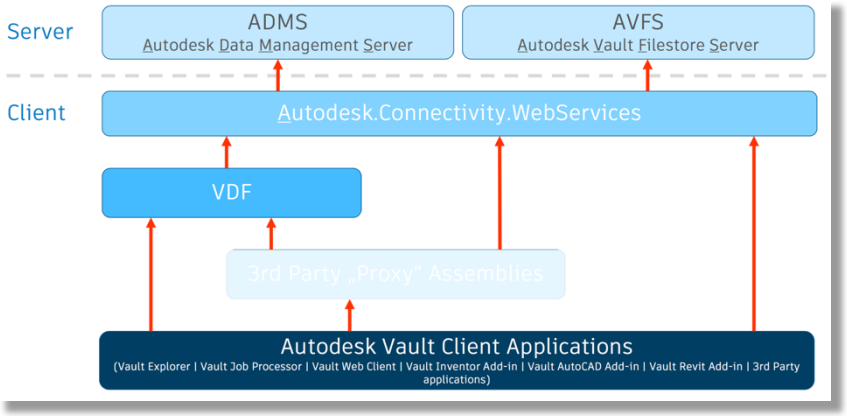
2 Understand the details

2.1 Autodesk Vault API Overview

Vault is a client/server architecture; the access to server components – ADMS (Autodesk Data Management Server) and AVFS (Autodesk Vault Filestore Server) leverages client side web services. This requires either a Vault Client installed or the re-use of SDK libraries for independent applications.

The Vault SDK libraries are composed using the Microsoft .NET Framework (version 4.5 for Vault 2018)

2.1.1 Architecture



There are two different levels of APIs that can be used to accomplish different tasks:

The ***Vault API*** consists of web services that contain all the different services and functions needed to interact with Vault. The *Vault API* is also known as *Autodesk.Connectivity.WebServices*.

The second level of API is provided by the ***Vault Development Framework (VDF)***. The *VDF* is a high-level framework on top of the existing API that provides:

- Reusable business logic for common vault algorithms
- Reusable GUI controls for common workflows
- Extensible components that can be configured to match host requirements.

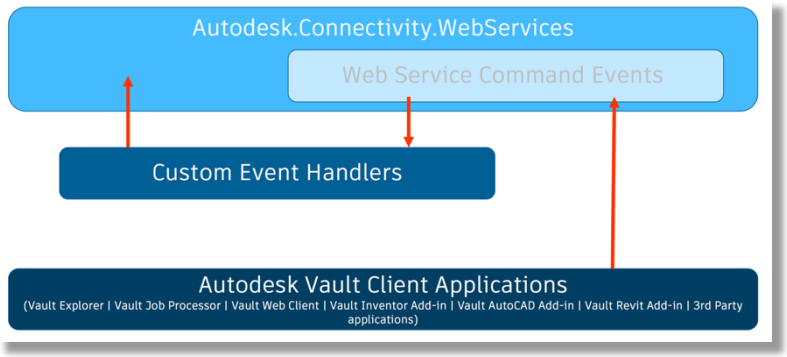
The *VDF* is documented online:

https://knowledge.autodesk.com/support/vault-products/learn-explore/caas/CloudHelp/cloudhelp/Help/ENU/Vault/files/GUID-1E65A09E-D9DC-42A8-871B-568004C43A56-htm.html

---

Note – If specific functionality is needed to be used repeatedly by different applications or extensions, one's own "proxy" assemblies can be created. Those proxies can reference the Vault APIs needed by the functionality and can be distributed with the different applications or extensions.

2.1.2 Event Handling

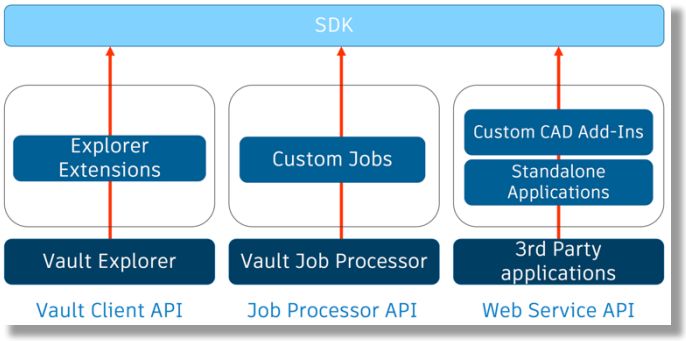


Command Events are part of the Vault API and can be subscribed by ***Custom Event Handlers***.

Whether it's a Vault Explorer Client, an Inventor Vault-AddIn or whether its one's own application; every single communication between a Vault Client and a Vault Server uses the Vault API and therefore *Custom Event Handlers* can subscribe to predefined actions, independent of the application that causes the event.

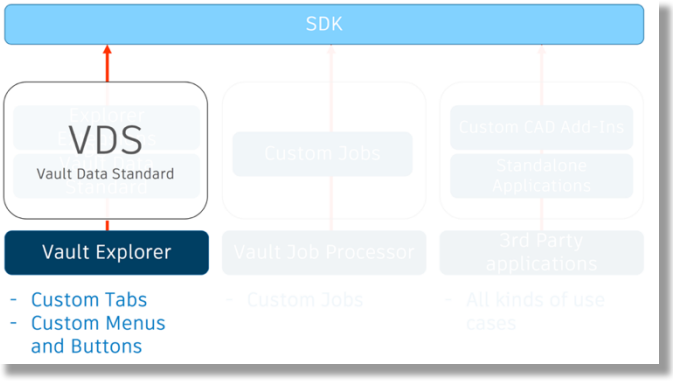
2.1.3 Extensibility

Different applications require different kinds of extensibility. That's why the SDK consist of three different APIs:



- ***Web Service API*** allows communication with the Autodesk Data Management Server (ADMS). Use this API any time you want to get or set server data.
- ***Vault Client API*** allows customization of the Vault client. Use this API to add custom commands and tabs to the UI.
- ***Job Processor API*** allows customization of the Job Processor client. Use this API to add handlers for custom job types in the job service queue.

2.1.4 Data Standard



Vault Data Standard is a data control feature for the Vault Client, Inventor, and AutoCAD that allows administrators to enforce how users enter Vault data.

The Vault part of VDS is a special *Explorer Extension* that allows administrators to add custom tabs, custom menus and menu buttons and custom functionality. All customization in VDS is done via scripting (PowerShell) or in XML files. The UI can be extended using *WPF (Windows Presentation Foundation)* based dialogs and controls. Therefore, the files distributed with Data Standard can be edited without special editors and there is no need to compile any of the distributed files.

Note – This class doesn't handle Vault Data Standard because one can use it to customize Vault without any programming knowledge or knowledge about the Vault APIs.
The AU class "SD124422 - Vault Extensions Snorkeling – First Touch to Vault Extension and Automation Programming" provides details about customization abilities of Vault Data Standard

2.2 Extension Basics

Vault extensions can be either ***Explorer Extensions***, ***Custom Jobs*** or ***Custom Event Handlers***. They all have in common, that they have to implement a specific interface, that they must deploy specific files and that they have to be copied to specific locations in order to become a Vault Extension.

2.2.1 Interfaces

An *Explorer Extension* implements the interface ***IExplorerExtension***, a *Custom Job* implements ***IJobHander*** and a *Custom Event Handler* implements ***IWebServiceExtension***. These interfaces are exposed by the following assemblies that are part of the Vault SDK:

- *Autodesk.Connectivity.Explorer.Extensibility.dll*
- *Autodesk.Connectivity.JobProcessor.Extensibility.dll*
- *Autodesk.Connectivity.Webservices.dll*

Different types of customizations can be combined in one assembly. Which Interface is implemented by the customization has to be configured in a file with extension *vcet.config*. This file specifies the interface and which class and assembly implements the interface.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectivity.ExtensionSettings3>
    <extension
      interface="Autodesk.Connectivity.Explorer.Extensibility.IExplorerExtension, Autodesk.Connectivity.Explorer.Extensibility, Versi
      type="MFG124959.LinkProperties.ExplorerExtension.ExplorerExtension, MFG124959.LinkProperties.ExplorerExtension">
    </extension>
  </connectivity.ExtensionSettings3>
</configuration>
```

2.2.2 Attributes

All Vault Extensions must contain specific assembly attributes to be recognized as an extension:

```
[assembly: AssemblyTitle("MFG124959.LinkProperties.ExplorerExtension")]
[assembly: AssemblyDescription("AU 2017 Example Vault Explorer Extension")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("coolOrange S.r.l.")]
[assembly: AssemblyProduct("MFG124959.LinkProperties.ExplorerExtension")]
[assembly: AssemblyCopyright("Copyright © coolOrange S.r.l. 2017")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

The assembly attributes ***AssemblyCompany***, ***AssemblyProduct*** and ***AssemblyDescription*** must not be empty. The Framework attributes (*Autodesk.Connectivity.Extensibility.Framework*) ***ExtensionId*** must contain a unique GUID that identifies the extension and the ***ApiVersion*** must contain the internal version of Vault (11.0 for Vault 2018)

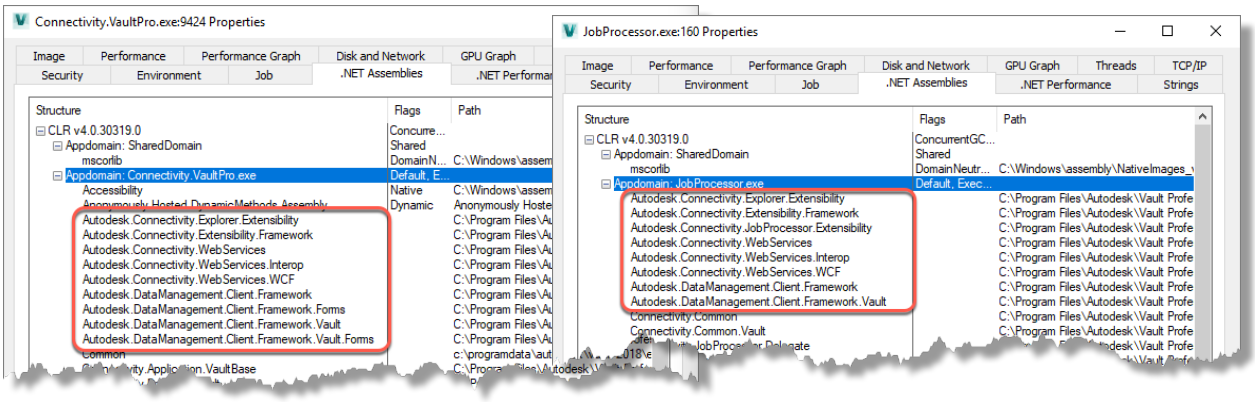
```
[assembly: Autodesk.Connectivity.Extensibility.Framework.ExtensionId("7ADC0766-F085-46d7-A2EB-C68F79CBF4E1")]
[assembly: Autodesk.Connectivity.Extensibility.Framework.ApiVersion("11.0")]
```

2.2.3 Deployment

All extensions have to reside in a directory underneath the Vault Extensions folder: *C:\ProgramData\Autodesk\Vault 2018\Extensions*

The configuration file (*vcet.config*), the assembly that implements the interface(s) and all referenced assemblies have to be deployed to that folder.

Assemblies, loaded by the host application don't have to be deployed.



2.2.4 Error handling

Explorer Extensions are loaded by the Vault Explorer at startup, ***Custom Job Handlers*** are loaded by the JobProcessor on startup and ***Custom Event Handlers*** are loaded by any application that is using the Vault API at the startup of that application.

If – for some reason – the Vault client cannot load the extension, an Extension Loading Error Logging can be activated by creating the file *C:\ProgramData\Autodesk\Vault 2018\Extensions\ExtensionLogConfig.xml*

```
<?xml version="1.0"?>
<ExtensionLoggerConfiguration>
  <BaseFileName>C:\ProgramData\Autodesk\Vault 2018\Extensions\ExtensionLog_</BaseFileName>
  <DateFormat>_yyyyMMdd</DateFormat>
  <KeepLogsFor>1</KeepLogsFor>
  <FileExtension>.txt</FileExtension>
</ExtensionLoggerConfiguration>
```

Once the XML file is present, an error logging get executed when an Extension gets loaded. In the XML file, the *file name*, the *location*, the *date format*, the *lifespan* and *file extension* of the log file is configurable.

More detailed information about Extension Loading Error Handling can be found online:
http://justonesandzeros.typepad.com/blog/2012/07/extension-loading-error-logging.html

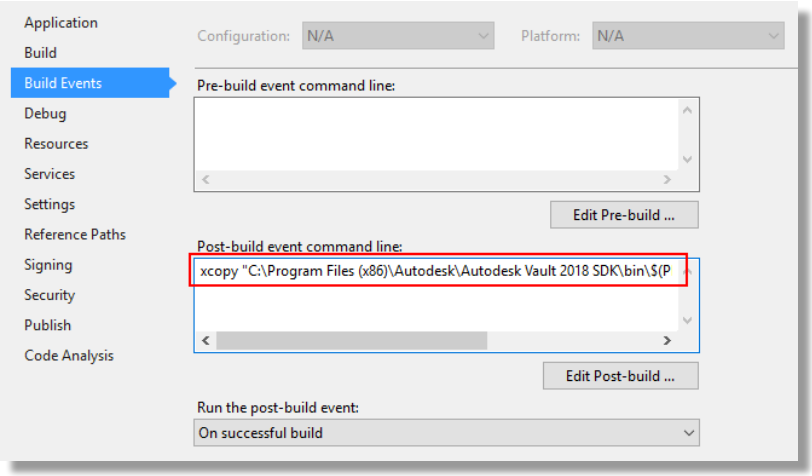
2.3 Standalone Application Basics

A standalone application can utilize the *Vault API* and – if needed – the *VDF* to interact with Vault. Unlike a Vault Extension, a standalone application is not hosted by an existing Vault client and therefore needs to deploy all the referenced assemblies.

2.3.1 Deployment

All standalone applications communicating with Vault must deploy the following assemblies:

- *clmloader.dll*
- *Autodesk.Connectivity.WebServices.dll*
- *Autodesk.Connectivity.WebServices.Interop.dll*
- *Autodesk.Connectivity.WebServices.WCF.dll*



clmloader.dll is not a managed dll it cannot be deployed the same way a referenced assembly is deployed in the Visual Studio project.

Instead, it can be deployed using ***xcopy*** as a post-build event command:

xcopy "C:\Program Files (x86)\Autodesk\Autodesk Vault 2018 SDK\bin\$(PlatformName)\clmloader.dll" "$(TargetDir)" /y

2.4 Licensing

Vault 2017 and higher versions are using the licensing framework *CLiC* that became part to the Vault SDK.

2.4.1 CLiC

To allow the Vault API to check if a valid license is avaliable, the following componentes are needed and obligatory for standalone applications:

- *clmloader.dll*
- *Autodesk.Connectivity.WebServices.Interop.dll*

The Vault edition that have to be requested by the licensing framework can be configured in the *App.config* file of one's standalone application by adding a configuration section of type *Autodesk.Connectivity.WebServices.LicensingSection*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <configSections>
    <section name="Licensing" type="Autodesk.Connectivity.WebServices.LicensingSection, Autodesk.Connectivity.WebServices"/>
  </configSections>

  <Licensing edition="Professional"/>

</configuration>
```

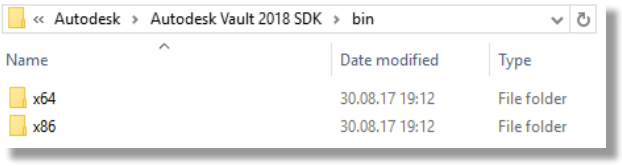
Note – For Vault Extensions, licensing doesn't have to be taken into account, because all components are already loaded by the host applications

2.4.2 Bitness (32-bit vs. 64-bit)

The *Vault SDK* includes x64 and x86 versions of all DLLs. Depending on the platform of the host application, the appropriate versions have to be referenced.

Only Vault Office Client is available in x86, Vault Workgroup and Vault Professional extensions have to be created in x64.

Standalone applications can either be x86 or x64. Depending on the platform, the appropriate assemblies have to be referenced in the Visual Studio project.



3 Use Case 1: Make use of Inventor Document's Internal ID

3.1 Use Case 1 | Description

The internal ID of an Inventor document doesn't change when a document gets duplicated. Unfortunately, Vault doesn't store this information, otherwise one could use the internal ID to search for similar components without the need to analyze the geometry of the Inventor document.

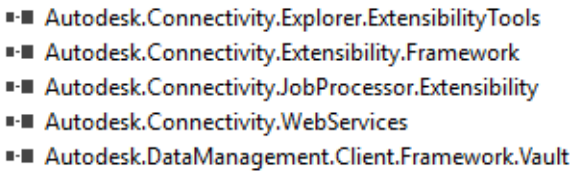
This use case is all about to get the internal ID of an Inventor document.

3.2 Use Case 1 | Custom Job Handler Example

In the first example, a job needs to be created that searches the Vault for all Inventor documents where the *User Defined Property* "Internal ID" is empty. It downloads and opens the file, retrieves the internal ID and updates the *UPD* with that information.

3.2.1 Create a Custom Job

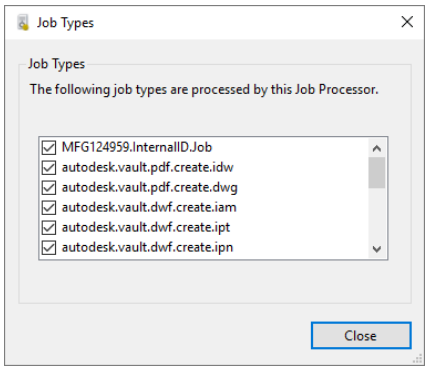
The Visual Studio solution ***MFG124959 - Vault Extensions Deep Dive*** contains the sample project ***MFG124959.InternalID.Job*** that one can use as a reference for the creation of a *Custom Job*. This project contains references to the following assemblies located in the bin directory of the Vault SDK (x64):



The class *JobHandler* is present and implements the interface *IJobHandler* as well as the member functions of that interface.
Furthermore, a *vcet.config* file is present which option *Copy to Output Directory* is set to *Copy always.*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectivity.ExtensionSettings3>
    <extension
       interface="Autodesk.Connectivity.JobProcessor.Extensibility.IJobHandler,
          Autodesk.Connectivity.JobProcessor.Extensibility, Version=23.0.0.0, Culture=neutral,
          PublicKeyToken=aa20f34aedd220e1"
       type="MFG124959.InternalID.Job.JobHandler, MFG124959.InternalID.Job">
      <setting key="JobType1" value="MFG124959.InternalID.Job"/>
    </extension>
  </connectivity.ExtensionSettings3>
</configuration>
```

Note – One Custom Job Handler assembly can contain many different Job Types. The Job Type of a Custom Job Handler is only added to the JobProcessor's list of Job Types, if the particular job is mentioned in the vcet.config as a <setting/> node



3.2.2 Debug a Custom Job

From the project ***MFG124959.InternalID.Job*** can also be obtained how debugging can be applied to a *Custom Job*. The output of the project is set to a subfolder of the Extension directory: *C:\ProgramData\Autodesk\Vault 2018\Extensions\MFG124959.InternalID.Job*

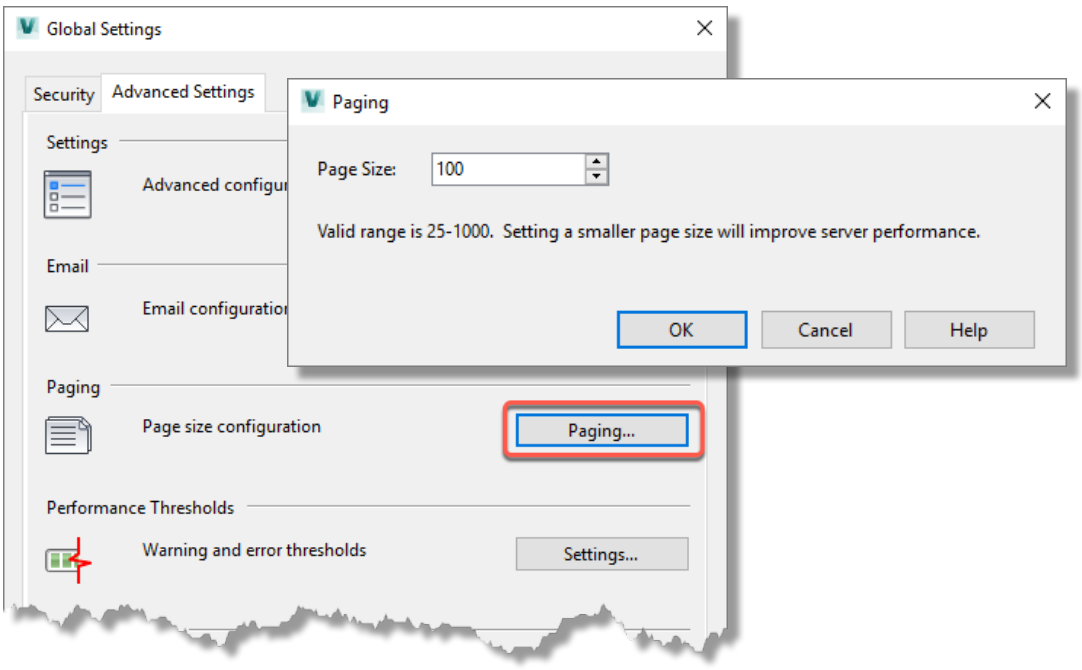
The Start Action is set to *Start external program* using the argument *C:\Program Files\Autodesk\Vault Professional 2018\Explorer\Connectivity.JobProcessor.Delegate.Host.exe*

3.2.3 Find entities in Vault (Paging)

To find all files with an empty *UDP* "Internal ID" the property definition of that *UDP* needs to be determined from all available property definitions and with that information a new search condition needs to be created. One or more search Conditions can be passed to the service method "*FindFilesBySearchConditions*" of the *DocumentService*:

```
string bookmark = null;
SrchStatus status = null;
var files = new List<File>();
while (status == null || files.Count < status.TotalHits)
{
    var results = wsm.DocumentService.FindFilesBySearchConditions(
        new[] { srchCondInternalId, srchCondInternalProvider, srchCondDateTime },
        null,
        null,
        false,
        true,
        ref bookmark,
        out status);

    if (results != null)
        files.AddRange(results);
    else
break;
}
```

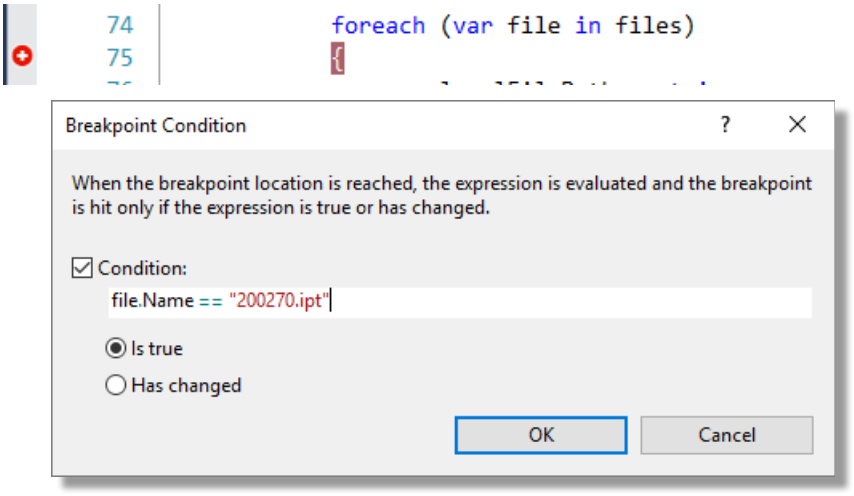
The interesting thing is that this service method doesn't return all entities if the result is bigger than the page size configured in the *ADMS Console*. In this case, *FindFilesBySearchConditions* have to be called in a loop as long as all results are returned for a particular search.



Note – You can use Telerik's Fiddler extended by coolOrange's vapiTrace to record all Vault API calls of manual performed tasks in Vault. For installation and further information visit the acoording webpages of Telerik, http://www.telerik.com/fiddler, and coolOrange, https://www.coolorange.com/en/apps_vault.php. A video tutorial in Vault Knowledge Network instructs how to configure and use this powerful tool with the sample adding a folder with project category in Vault

3.2.4 Conditional breakpoints

In some circumstances, a huge amount of data is returned by a Vault service method in form of an array of objects. During the debugging, one may only be interested in a particular element out of this array. Instead of looping through all the array elements until the interresting one gets hit, a conditional breakpoint is the most useful tool.



Conditional breakpoints can be activated in the contextual menu of a breakpoint. If a small, white icon is displayed in the middle of the breakpoint, a condition is set and active.

3.2.5 File download using the Vault API

Downloading a file using the *Vault API* looks complicated at first sight, but becomes easier with the use of a helper function that converts the stream that is returned by the service method *DownloadFilePart* to an array of bytes:

```
var fileSize = file.FileSize;
var maxPartSize = wsm.FilestoreService.GetMaximumPartSize();
var ticket = wsm.DocumentService.GetDownloadTicketsByMasterIds(new[] { file.MasterId })[0];
byte[] bytes;

using (var stream = new System.IO.MemoryStream())
{
    var startByte = 0;
    while (startByte < fileSize)
    {
        var endByte = startByte + maxPartSize;
        if (endByte > fileSize)
            endByte = fileSize;

        using (var filePart = wsm.FilestoreService.DownloadFilePart(
            ticket.Bytes, startByte, endByte, true))
        {
            byte[] buffer = StreamToByteArray(filePart);
            stream.Write(buffer, 0, buffer.Length);
            startByte += buffer.Length;
        }
    }

    bytes = new byte[stream.Length];
    stream.Seek(0, System.IO.SeekOrigin.Begin);
    stream.Read(bytes, 0, (int)stream.Length);

    stream.Close();
}

System.IO.File.WriteAllBytes(localFilePath, bytes);


private byte[] StreamToByteArray(System.IO.Stream stream)
{
    using (var memoryStream = new System.IO.MemoryStream())
    {
        stream.CopyTo(memoryStream);
        return memoryStream.ToArray();
    }
}
```

3.3 Use Case 1 | Standalone application Example

Now that a *Custom Job* is present that completes the Vault with the internal Inventor Document IDs, we have to execute this job – ideally periodically – to keep the information up-to-date.

Usually a job gets triggered by a status change of a Vault entity, but the *Vault API* can also be used to submit a job to Vaults job queue.

3.3.1 Assembly deployment

The Visual Studio solution ***MFG124959 - Vault Extensions Deep Dive*** contains the sample project ***MFG124959.Standalone.JobTrigger*** This project is a standalone application that triggers a job once it is launched. This project contains references to the following assemblies located in the bin directory of the Vault SDK (x64):

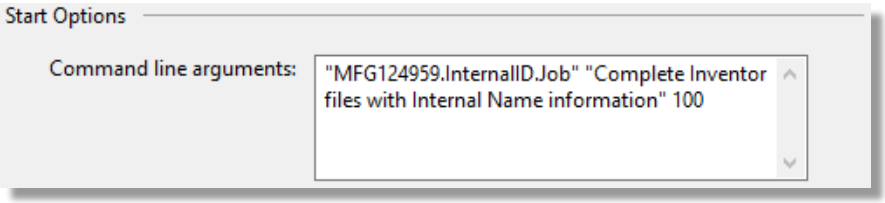
- Autodesk.Connectivity.WebServices
- Autodesk.Connectivity.WebServices.Interop
- Autodesk.Connectivity.WebServices.WCF

Furthermore, the post build action executes the following command:

xcopy "C:\Program Files (x86)\Autodesk\Autodesk Vault 2018 SDK\bin\$(PlatformName)\clmloader.dll" "$(TargetDir)" /y

3.3.2 Debug a standalone application

Standalone applications can be debugged like any other console application. The only unique setting in this particular example is the need of command line arguments. Without these arguments, the application is supposed to quit with an error message.




```
if (args.Length != 3)
{
    Console.WriteLine(
      "Please specify 'job type', 'job description' and 'job priority' as command line arguments!");
    return;
}
```

3.3.3 License consumption

The *App.config* of this standalone application not only contains a section that stores the login information to Vault but also a section that controls the type of licensing that is used to login to Vault.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="Licensing" type="Autodesk.Connectivity.WebServices.LicensingSection,
        Autodesk.Connectivity.WebServices"/>
    <sectionGroup name="Settings">
      <section name="Vault" type="System.Configuration.AppSettingsSection, System.Configuration,
          Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
    </sectionGroup>
  </configSections>

  <Licensing edition="None"/>

  <Settings>
    <Vault>
      <add key="DataServer" value="localhost"/>
      <add key="FileServer" value="localhost"/>
      <add key="Vault" value="MFG124959"/>
      <add key="Username" value="Administrator"/>
      <add key="Password" value=""/>
    </Vault>
  </Settings>

  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

</configuration>
```

Since this application only signs in to Vault, submits a Job and finally signs out from Vault again, there is no need to consume a license.

Note – To submit a job to Vaults Job Queue
it's not necessary to obtain a license

3.3.4 Login to Vault using the Vault API

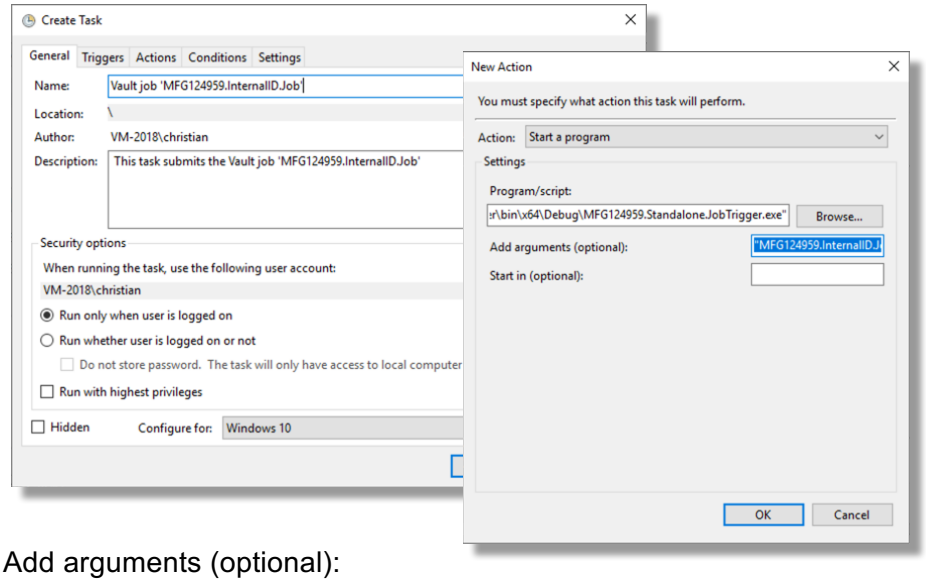
To sign in to Vault from within a standalone application and by using the *Vault API*, an object of type ***WebServiceManager*** has to be created. Once an object that implements the interface

IWebserviceCredentials is passed to the constructor of the *WebServiceManager* (in case of this example an object of type ***UserPasswordCredentials***), the login to Vault is executed. The *ReadOnly* parameter of the *UserPasswordCredentials* is set to *true*!

```
var identities = new ServerIdentities
{
    DataServer = dataServer, FileServer = fileServer
};
var userPasswordCridentials = new UserPasswordCredentials(
    identities, vault, username, password, true);
var wsm = new WebServiceManager(userPasswordCridentials);
```

3.3.5 Utilizing the Windows Task scheduler

To periodically launch the standalone application one can use the operating system utility "*Windows Task scheduler*" and pass all needed information to the standalone application by using command line arguments:



Add arguments (optional):

"MFG124959.InternalID.Job" "Complete Inventor files with Internal Name information" 100

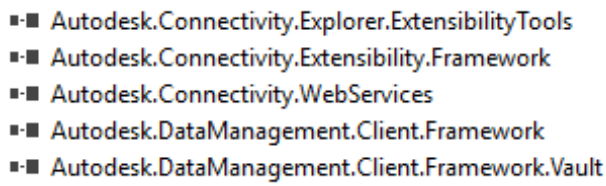
3.4 Use Case 1 | Custom Event Handler Example

The disadvantage of updating the file information periodically is, that it happens asynchronously which means the internal ID is not yet collected by the time the user needs the information. That's

where a events are useful: A *Custom Event Handler* can react whenever an Inventor file is added to Vault and can update the *UDP* automatically and synchronously.

3.4.1 Create a Custom Event Handler

The Visual Studio solution ***MFG124959 - Vault Extensions Deep Dive*** contains the sample project ***MFG124959.InternalID.EventHandler*** that one can use as a reference for the creation of a *Custom Event Handler*. This project contains references to the following assemblies located in the bin directory of the Vault SDK (x64):



The class *WebServiceExtension* is present and implements the interface *IWebServiceExtension* as well as the member functions of that interface.
Furthermore, a *vcet.config* file is present which option *Copy to Output Directory* is set to *Copy always.*

```
<configuration>
  <connectivity.ExtensionSettings3>
    <extension
      interface="Autodesk.Connectivity.WebServices.IWebServiceExtension,
           Autodesk.Connectivity.WebServices, Version=23.0.0.0, Culture=neutral,
           PublicKeyToken=aa20f34aedd220e1"
      type="MFG124959.InternalID.EventHandler.WebServiceExtension,
           MFG124959.InternalID.EventHandler">
    </extension>
  </connectivity.ExtensionSettings3>
</configuration>
```

3.4.2 Debug a Custom Event Handler

The project ***MFG124959.InternalID.EventHandler*** can also show how debugging can be applied to a *Custom Event Handler*. The output of the project is set to a subfolder of the Extension directory:

C:\ProgramData\Autodesk\Vault 2018\Extensions\MFG124959.InternalID.EventHandler

Since a *Custom Event Handler* can subscribe to events, idenpendent of the executing application, the *Start external program* can be set to any application that the is using the *Vault API*.

3.4.3 Detect the calling application

In order to recoginze the application that caused the event withing a *Custom Event Handler*, two functions from the .NET Framework can be utilized:

```
System.Reflection.Assembly.GetEntryAssembly()
```

and

```
System.Reflection.Assembly.GetCallingAssembly()
```

GetEntryAssembly() returns the managed (.NET) assembly of the application that was started by the user. Since Autodesk Inventor is not a .NET application, this function returns *null* in case of Inventor.

GetCallingAssembly() returns the *Autodesk.Connectivity.WebServices.dll* that was used to subscribe to the event, in case of Inventor to the one that is deployed with the Inventor Vault Add-In.

Using the information from these two functions combined with the string that is written to the comment of a file, one can detect whether the handled event was executed by Vault Explorer, Copy Design, Autoloader, Inventor, AutoCAD or a 3rd party standalone application.

3.4.4 Create a VDF connection

While the *Custom Job* example exclusively uses the *Vault API*, this example uses *VDF* (*Vault Development Framework*) to speed up developing on the one hand and to demonstrate the differences on the other hand.

Using the *VDF*, a ***Connection*** (*Autodesk.DataManagement.Client.Framework.Vault.Currency.Connections.Connection*) object can be created using selected properties of the ***IWebService*** representation:

```
using VDF = Autodesk.DataManagement.Client.Framework;
…

var service = sender as IWebService;
if (service == null)
    return;

var wsm = service.WebServiceManager;
var connection = new VDF.Vault.Currency.Connections.Connection(
    wsm,
    wsm.WebServiceCredentials.VaultName,
    service.SecurityHeader.UserId,
    wsm.WebServiceCredentials.ServerIdentities.DataServer,
    VDF.Vault.Currency.Connections.AuthenticationFlags.Standard);
```

The connection object is needed by most of the *VDF* functionalities.

3.4.5 File download using the VDF

While the download of a file using the *Vault API* is complex, even with the use of helpful functions, the download using the *VDF* is way easier:

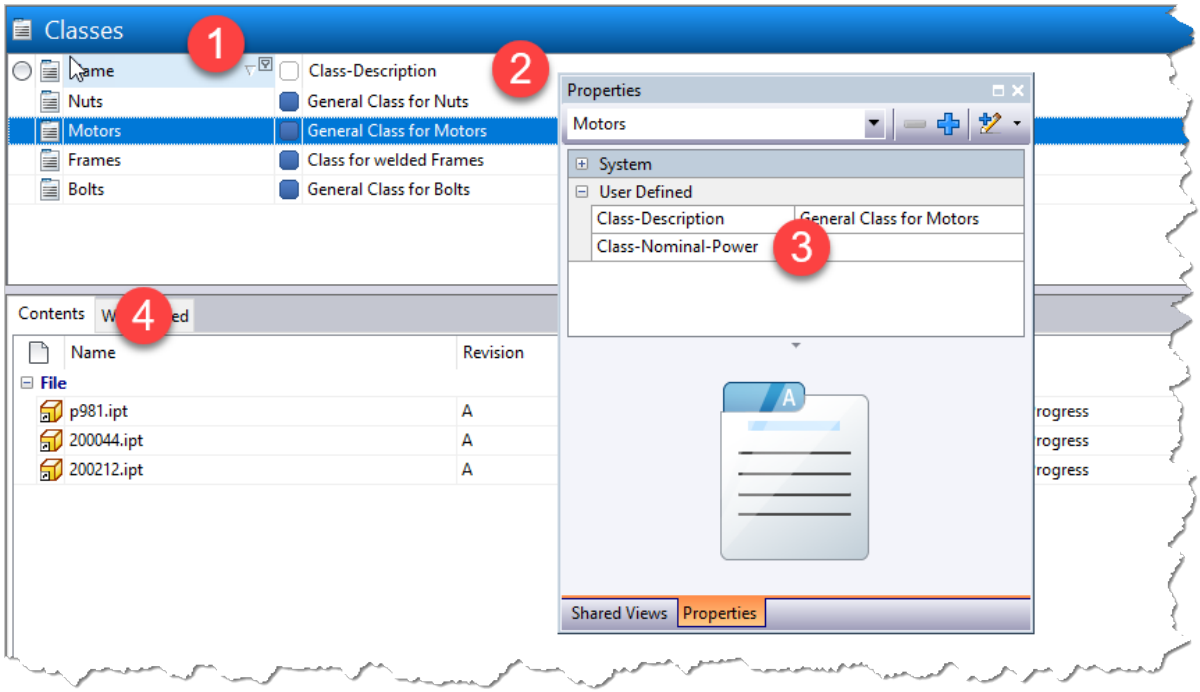
```
var fileIteration = new VDF.Vault.Currency.Entities.FileIteration(connection, file);

var folderPathAbsolute = new VDF.Currency.FolderPathAbsolute(@"C:\temp\");
var acquireSettings = new VDF.Vault.Settings.AcquireFilesSettings(connection)
{
    DefaultAcquisitionOption = VDF.Vault.Settings.AcquireFilesSettings.AcquisitionOption.Download,
    LocalPath = folderPathAbsolute
};
acquireSettings.AddEntityToAcquire(fileIteration);

var acquireFiles = connection.FileManager.AcquireFiles(acquireSettings);
var fileResult = acquireFiles.FileResults.First();
localFilePath = fileResult.LocalPath.FullPath;
```

A special file object needs to be created (***FileIteration***), a download options object has to be created and filled with options and the file(s) to download and finally a single download function has to be called (***AcquireFiles***).

4 Use Case 2: Select a Custom Object from within Inventor

4.1 Use Case 2 | Description

For component classification purposes the custom object “Class” is configured in Vault (1). Each class shares common properties like “Class-Description” (2) and individual ones, e.g., Class-Nominal-Power for Motors (3).



Files can be linked to this class to reflect being a member of this class (4). Vault Explorer allows linking using copy & paste commands. Unfortunately, an engineer cannot select a “Class” while working in Inventor using any out-of-the-box mechanism. Therefore, a custom Inventor Add-In has to be created to enable a workflow like this.

This use case demonstrates how to connect to Vault, search custom objects of type “Class” and displays those classes in a grid within Inventor. Selecting a class, the Add-In writes its name into the document’s iProperty. Of course, for “real” classification you need to continue, creating links or adding the class’ individual properties to the Inventor file. Whatever your particular workflow requires in detail, this sample shares the primary starting point to access the classification objects within Inventor.

4.2 Use Case 2 | Inventor Add-In Example

The creation as well as the debugging mechanism of an Inventor Add-In are way too complex to handle it along the specialties of Vault Extensions. Detailed documentation can be obtained from the online documentation:

http://help.autodesk.com/view/INVNTOR/2018/ENU/?guid=GUID-52422162-1784-4E8F-B495-CDB7BE9987AB

4.2.1 Reuse Vault connection from Inventor's Vault Add-In

In an Inventor Add-In that is communicating with Vault, the user doesn't want to login to Vault when he or she is already signed in by Inventor's Vault Add-In. Therefore, this example shows how to use *Reflection* to obtain the existing connection and bring up the login dialog that is integrated to the Inventor Vault Add-In if the connection doesn't already exist:

```
var fileName = System.Diagnostics.Process.GetCurrentProcess().MainModule.FileName;
var directoryName = Path.GetDirectoryName(fileName);
if (directoryName == null)
    return null;

var edmAddinDll = Path.Combine(directoryName, "Connectivity.InventorAddin.EdmAddin.dll");

if (!File.Exists(edmAddinDll))
{
    Console.WriteLine(@"EdmAddin not installed!");
    return null;
}

var assembly = System.Reflection.Assembly.LoadFrom(edmAddinDll);
var edmSecurity = assembly.GetTypes().First(c => c.Name.Contains("EdmSecurity"));

var propInstance = edmSecurity.GetProperty(
    "Instance",
    System.Reflection.BindingFlags.Static | System.Reflection.BindingFlags.Public);
var instance = propInstance.GetValue(null, null);

var methodInfo = instance.GetType().GetMethod("IsSignedIn");
var isSignedIn = methodInfo.Invoke(instance, null);
if (isSignedIn == null || Convert.ToBoolean(isSignedIn) != true)
    application.CommandManager.ControlDefinitions["LoginCmdIntName"].Execute();

var propVaultConnection = instance.GetType()
    .GetProperty(
        "VaultConnection",
        System.Reflection.BindingFlags.Instance |
        System.Reflection.BindingFlags.Public);

var vaultConnection = propVaultConnection.GetValue(instance, null);
```

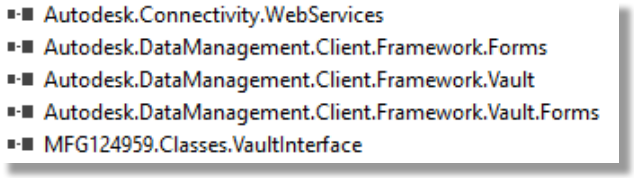
4.2.2 Late binding to create one Inventor Add-In for multiple Vault versions

Autodesk Inventor's API doesn't change very often and most of the Inventor Add-Ins can be used with multiple versions. Whereas every Vault version brings out new and dedicated Vault APIs that are only available for this particular Vault version.

To reuse the Inventor Add-In in different Inventor versions and only replace the "Vault part" of the Add-In, this example uses a techique called "Late binding" to load the DLL - that contains the *Vault API* calls - at runtime. To accomplish this, an own assembly with an interface is needed that defines all the functions that are used by the Inventor Add-In:

```
namespace MFG124959.Classes.VaultInterface
{
    public interface IVaultFunctions
    {
        void Initialize(object connection);
        string GetCustomEntityValue(string custEntDefName);
    }
}
```

Besides that, for each Vault version an additional assembly is necessary that implements the interface and references the Vault API assemblies needed for this particular Vault version.




```
public class VaultFunctions : IVaultFunctions
{
    private VDF.Vault.Currency.Connections.Connection _connection;
    private WebServiceManager _wsm;
    private CustEntDef[] _custEntDefs;

    public void Initialize(object connection)
    {
        _connection = (VDF.Vault.Currency.Connections.Connection)connection;
        _wsm = _connection.WebServiceManager;
        _custEntDefs = _wsm.CustomEntityService.GetAllCustomEntityDefinitions();
    }

    public string GetCustomEntityValue(string custEntDefName)
    {
        var entity = SelectCustomEntity(custEntDefName);
      if (entity != null)
            return entity.EntityName;

        return null;
    }
…
```

Finally, the Inventor Add-In needs a reference to the assembly that defines the interface and the assembly loader functionality in the example Add-In is using the assembly, that is configured in the App.config of the Add-In.

```
<Settings>
  <Vault>
    <add key="ImplementationAssembly" value="MFG124959.Classes.Vault2018.dll"/>
  </Vault>
</Settings>
```

4.2.3 Create an object picker dialog using a VDF grid for custom objects

Now that the Vault functionality is outsourced and the Vault Add-in is independent of the Vault version, one has to implement the functionality in the external assembly. The example **MFG124959.Classes.Vault2018** implements the interface and provides a dialog that is shown to select a “Class” which is basically a custom object.

The example uses *VDF* functionality but since there comes a built in Folder picker dialog and a File picker dialog but not a custom object picker dialog, one has to create an own dialog. The example provides such a dialog composed by a *WinForms* Dialog and a *VaultBrowserControl* (*Autodesk.DataManagement.Client.Framework.Vault.Forms.Controls.VaultBrowserControl*).

When the *VaultBrowserControl* is placed to the *Form*, data can be queried, using the *Vault API* service method *FindCustomEntitiesBySearchConditions* from the *CustomEntityService*.

```
string bookmark = null;
SrchStatus status = null;
while (status == null || custEnts.Count < status.TotalHits)
{
    var results = _wsm.CustomEntityService.FindCustomEntitiesBySearchConditions(
    new[] { srcCond }, null, ref bookmark, out status);

    if (results != null)
        custEnts.AddRange(results);
   else
       break;
}
```

Note – Similar to FindFilesBySearchConditions the page size has to be considered when retrieving custom objects using the service method FindCustomEntitiesBySearchConditions

Finally, all the custom objecst have to be converted to *VDF* entities of type *CustomObject*:

```
foreach (var custEnt in custEnts)
   entities.Add(new VDF.Vault.Currency.Entities.CustomObject(_connection, custEnt));
```

and set as data source to the *VDF* grid (along with configuration options that are defining the grids appearance)

```
private readonly VDF.Vault.Forms.Models.ViewVaultNavigationModel _navigationModel;

…

var configuration = new VDF.Vault.Forms.Controls.VaultBrowserControl.Configuration(

    conn, persistenceKey, null);
configuration.AddInitialColumn("Name");
configuration.AddInitialSortCriteria("Name", true);

_navigationModel = new VDF.Vault.Forms.Models.ViewVaultNavigationModel();
_navigationModel.AddContent(entities);

vaultBrowserControl1.SetDataSource(configuration, _navigationModel);
```

4.3 Use Case 2 | Custom Event Handler Example

Now that the Inventor Add-In write the “Class” name to an iProperty, the *Custom Event Handler* from the previous use case can be extended to automatically link the file to the custom object “Class” when the file is checked in to Vault the first time.

4.3.1 Link file to class

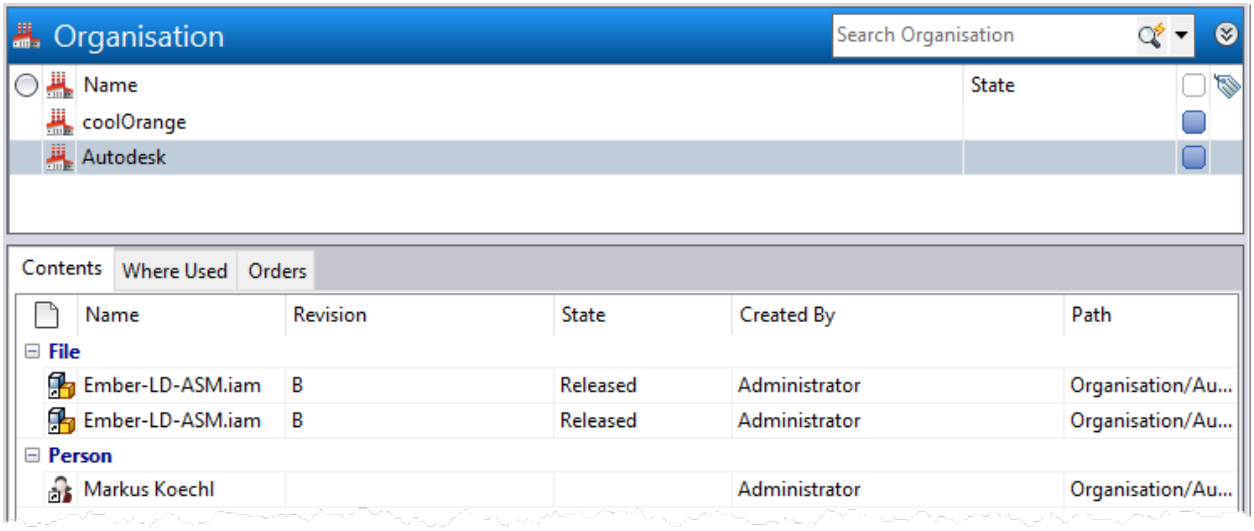
The existing *Custom Event Handler* created in use case 1 can be extended by reading out the iProperty when the file is already open to retrieve the Internal ID. Finally, the custom object can be searched by name (using *FindCustomEntitiesBySearchConditions* again with different search conditions) and a link between the custom object that is found by the search and the file that is processed by the *Custom Event Handler* can be created using the *Vault API*:

```
wsm.DocumentService.AddLink(custEnt.Id, "FILE", file.Id, null);
```

5 Use Case 3: Use Link Properties to track an Order history

5.1 Use Case 3 | Description

The entities of the custom object “Organisation” can be used to track how often a product was purchased by a particular customer. Therefore, the file can be linked with the one “Organisation” that represents the customer. Unfortunately, a link to a file always represents the relationship between the latest version of a file and another entity and therefore the version information is lost:



This use case is trying to solve this issue with an *Explorer Extension* that is capable to not only create the link between the file and the custom object but also saves *Link Properties* to the file that stores the version information and the order number.

To display all this information the *Explorer Extension* also needs to expose a custom tab in the context of a custom object with a grid displaying the file version information and the order number stored by the *Link Properties*.

Note – “Organisation” == Organization

5.2 Use Case 3 | Explorer Extension

5.2.1 Create Vault Explorer Extension

The Visual Studio solution ***MFG124959 - Vault Extensions Deep Dive*** contains the sample project ***MFG124959.LinkProperties.ExplorerExtension*** that one can use as a reference for the creation of a *Explorer Extension*. This project contains references to the following assemblies located in the bin directory of the Vault SDK (x64):

- Autodesk.Connectivity.Explorer.Extensibility
- Autodesk.Connectivity.Extensibility.Framework
- Autodesk.Connectivity.WebServices
- Autodesk.DataManagement.Client.Framework.Forms
- Autodesk.DataManagement.Client.Framework.Vault
- Autodesk.DataManagement.Client.Framework.Vault.Forms

The class *ExplorerExtension* is present and implements the interface *IExplorerExtension* as well as the member functions of that interface.
Furthermore, a *vcet.config* file is present which option *Copy to Output Directory* is set to *Copy always.*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectivity.ExtensionSettings3>
    <extension
      interface="Autodesk.Connectivity.Explorer.Extensibility.IExplorerExtension,
           Autodesk.Connectivity.Explorer.Extensibility, Version=23.0.0.0, Culture=neutral,
           PublicKeyToken=aa20f34aedd220e1"
      type="MFG124959.LinkProperties.ExplorerExtension.ExplorerExtension,
           MFG124959.LinkProperties.ExplorerExtension">
    </extension>
  </connectivity.ExtensionSettings3>
</configuration>
```

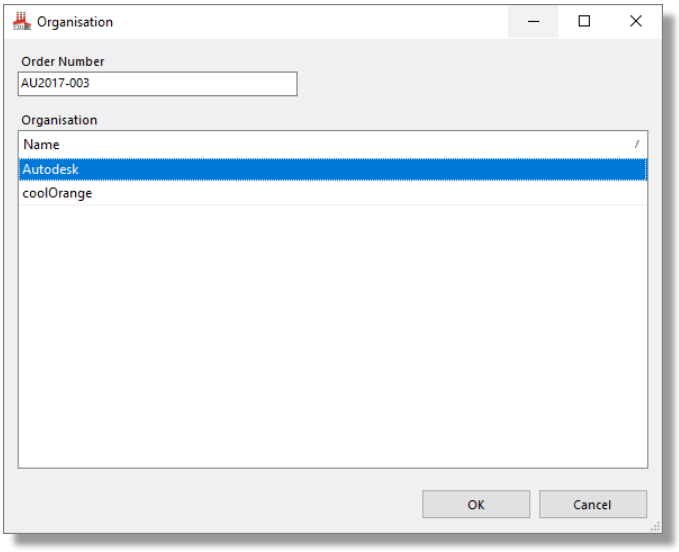
5.2.2 Debug Vault Explorer Extension

From the project ***MFG124959.LinkProperties.ExplorerExtension*** can also be obtained how debugging can be applied to an *Explorer Extension*. The output of the project is set to a subfolder of the *Extension* directory:

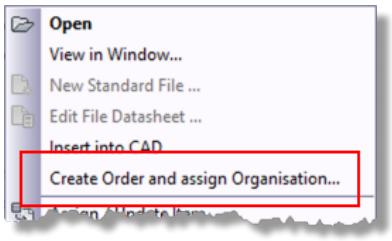
C:\ProgramData\Autodesk\Vault 2018\Extensions
MFG124959.LinkProperties.ExplorerExtension

The Start Action is set to *Start external program* using the argument *C:\Program Files\Autodesk\Vault Professional 2018\Explorer\Connectivity.VaultPro.exe*

5.2.3 Add custom command to link file to custom entity “Organisation”



Analog to the dialog grid in use case 2, the *Explorer Extension* includes a dialog where custom objects of type “Organisation” can be chosen. In addition, a string for the “Order Number” can be entered in this dialog.



The dialog is shown when the contextual menu entry “*Create Order and assign Organisation…*” of a file iteration in the History tab is clicked.

5.2.4 Create Link Properties

If this dialog is confirmed, the file is linked to the selected custom object and the file version and the entered “Order Number” are written to a Linked Properties.

Note – Link Properties are introduced to Vault 2018 and not available to previous versions

---

5.2.5 Create custom tabs on custom entities

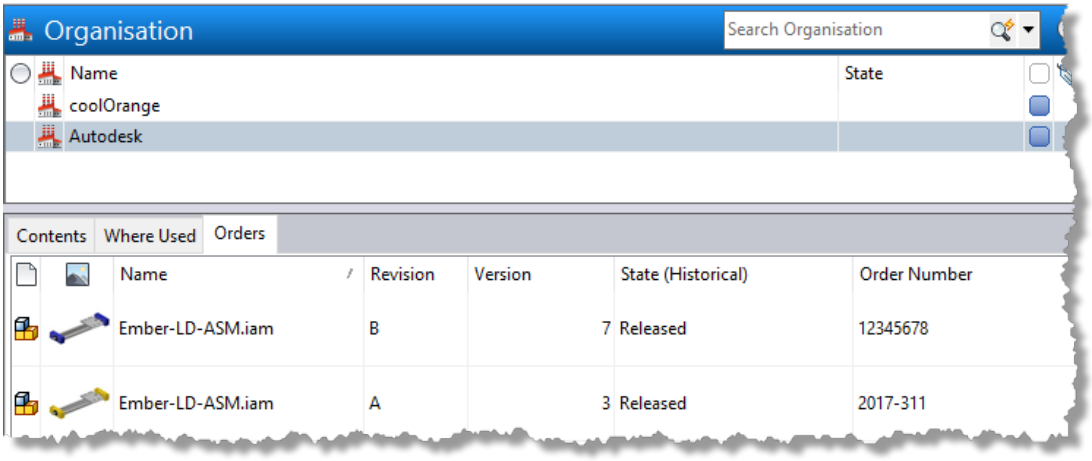
To add a custom tab to entities like files or folders, the *Vault API* provides the ***SelectionTypeId*** class with public members for files, folders and the like. For custom objects, there is no public member that represents the custom object because an instance of a custom object is not predefined but can be configured at runtime. Therefore, the *SelectionTypeId* constructor can be used to create a representation for a specific instance, e.g. "*Organisation*". The constructor expects the internal name of the custom object definition as parameter; unfortunately, this information is not available at the time a tab is loaded by the *Explorer Extension*.

Vault Data Standard faces the same problem and solves it by writing the internal name of each property definition to a JSON file. The example demonstrates how to use the information stored in this JSON file to create a user defined tab.

```
public IEnumerable<DetailPaneTab> DetailTabs()
{
    // The json file only gets created, if Vault DataStandard is installed
    var jsonFile = @"C:\ProgramData\Autodesk\Vault 2018\Extensions\DataStandard\" +
        @"Vault\CustomEntityDefinitions.json";
    if (System.IO.File.Exists(jsonFile))
    {
        var text = System.IO.File.ReadAllText(jsonFile);
        var definitions = Newtonsoft.Json.JsonConvert.
            DeserializeObject<CustomEntityDefinition[]>(text);
        foreach (var definition in definitions)
        {
            var entityDefinition = definition.EntityDefinitions.FirstOrDefault(
                e => e.dispNameField == "Organisation");
            if (entityDefinition != null)
            {
                var selectionTypeId = new SelectionTypeId(entityDefinition.nameField);
                var detailPaneTab = new DetailPaneTab(
                    "Organisation.Tab.OrdersTab",
                    "Orders",
                    selectionTypeId,
                    typeof(OrdersUserControl));
                detailPaneTab.SelectionChanged += OrganisationSelectionChanged;

                return new List<DetailPaneTab> { detailPaneTab };
            }
        }
    }

    return null;
}
```

5.2.6 Display VDF grid on a custom Vault Explorer tab

By using the same mechanisms as in use case 2 a *VaultBrowserControl* is placed to an *UserControl*. In the custom tab, all the entities linked to the selected custom object are gathered and the version information from the linked property is used the obtain the file with the correct version.



5.2.7 Extend the VDF grid to show the Link Property

The *VDF* offers the possibility to add columns to a grid, even if the entities that are shown don't contain that property. The example uses this functionality to show the "Order Number" that is stored in the *Link Property*, not in the link itself and not in the file iteration that is shown in the grid.

For that reason, the class *OrderPropertyExtensionProvider* is defined that implements the interface ***IPropertyExtensionProvider***. With the help of this provider the content of the additional column get refreshed every time the selection of the custom object changes.

```
public void Reload(VDF.Vault.Currency.Connections.Connection conn,
    IEnumerable<VDF.Vault.Currency.Entities.IEntity> entities,
    long propDefIdOrderNumber, long propDefIdOrderFileId, PropInst[] propInsts)
{
    var configuration = new VDF.Vault.Forms.Controls.VaultBrowserControl.Configuration(
        conn, "Grid.OrganisationOrder.LinkProperties", null);

    configuration.AddInitialColumn("Name");
    configuration.AddInitialSortCriteria("Name", true);

    var orderPropertyExtensionProvider = new OrderPropertyExtensionProvider(
        propDefIdOrderNumber,
        propDefIdOrderFileId,
        propInsts);
    configuration.AddPropertyExtensionProvider(orderPropertyExtensionProvider);

    _navigationModel = new VDF.Vault.Forms.Models.ViewVaultNavigationModel();
    _navigationModel.AddContent(entities);

    vaultBrowserControl1.SetDataSource(configuration, _navigationModel);
}
```

6 Additional resources

6.1.1 Vault

Vault API Blog:

http://justonesandzeros.typepad.com/

coolOrange Blog:

https://blog.coolorange.com/

Autodesk Knowledge Network:

https://knowledge.autodesk.com/support/vault-products

Manufacturing DevBlog:

http://adndevblog.typepad.com/manufacturing/

Vault customization forum:

http://forums.autodesk.com/t5/vault-customization/bd-p/301